

Reconfigurable State Machine Replication from Non-Reconfigurable Building Blocks

Vita Bortnikov¹, Gregory Chockler¹, Dmitri Perelman², Alexey Roytman¹, Shlomit Shachor¹, and
Ilya Shnayderman¹

¹IBM Research

²Technion, Israel Institute of Technology

Abstract

Reconfigurable state machine replication is an important enabler of elasticity for replicated cloud services, which must be able to dynamically adjust their size as a function of changing load and resource availability. We introduce a new *generic framework* to allow the reconfigurable state machine implementation to be derived from a collection of arbitrary non-reconfigurable state machines. Our reduction framework follows the *black box* approach, and does not make any assumptions with respect to its execution environment apart from reliable channels. It allows higher-level services to leverage *speculative command execution* to ensure uninterrupted progress during the reconfiguration periods as well as in situations where failures prevent the reconfiguration agreement from being reached in a timely fashion. We apply our framework to obtain a reconfigurable speculative state machine from the non-reconfigurable Paxos implementation, and analyze its performance on a realistic distributed testbed. Our results show that our framework incurs negligible overheads in the absence of reconfiguration, and allows steady throughput to be maintained throughout the reconfiguration periods.

Eligible for the best student paper award: Dmitri Perelman is a full-time graduate student

The paper includes **12.5** pages out of which approximately **2** are occupied by the figures.

1 Introduction

Replicated state machine, or RSM [11], is an important tool for maintaining integrity of distributed applications and services in failure-prone data center and cloud computing environments. In these settings, the infrastructure needs to adapt to changing resource availability, load fluctuations, variable power consumption, and data locality constraints. In order to meet these requirements, RSM must support *reconfiguration*, i.e., dynamic changes to replica set, or quorum system. It is essential to ensure that reconfiguration incurs minimum disruption to availability and performance, in order to enable building truly elastic services. The ability to perform reconfigurations in a non-disruptive fashion provides system designers with a powerful paradigm that can enable many optimizations. This includes proactive replacement of suspected or slow nodes at low cost, adapting to the changing environment characteristics (e.g. network delay, or diurnal load fluctuations, and many others).

Reconfigurable RSM has been proposed in multiple contexts (e.g., [14, 15, 10]). Each solution implemented a slightly different set of requirements, and all proved nontrivial. Designing this functionality for a realistic environment with minimal impact on performance is even more challenging. Naïve constructions follow the “brick-wall approach” in which the flow of user commands is stalled until the new configuration is installed and the state is transferred to it. Ideally, systems should strive to avoid this, and favor implementations with near-seamless hand-on, that maintain steady throughput and latency during the transition periods.

This paper introduces a framework for constructing reconfigurable state machines from collections of non-reconfigurable ones. We follow the *black box* approach that assumes nothing about the execution environment except the existence of reliable communication channels. Our reduction is both simple and generic, i.e., the underlying RSM implementation is completely opaque to the framework. Furthermore, it does not compromise efficiency, incurring negligible overhead in the absence of reconfigurations and avoiding system stalls upon reconfiguration.

The main ideas underlying our framework are as follows. Each newly proposed configuration is associated with its own instance of RSM, and all active RSM are executed concurrently to each other. The globally consistent *trunk* of commands is created by gluing together the totally ordered command sequences produced by each RSM. When switching from one RSM to another, the latter is chosen based on the outcome of the configuration agreement in the former. Our framework also relieves RSMs of state transfer responsibilities by ensuring that the latest trunk is transferred to the new configuration concurrently with the RSM execution. This way, each newly created RSM is completely independent from its predecessor, and in particular, can start executing from its initial state. We leverage this capability in our Paxos-based reconfigurable RSM implementation to supply each newly created Paxos instance with the identifier of a deterministically chosen leader thus eschewing the first phase of Paxos if the configured leader does not fail.

Another important optimization made possible by the RSM independence is the ability to *speculatively* overlap their execution with the reconfiguration protocol, thus considerably reducing the command latency during reconfiguration periods. Specifically, each RSM is made available for accepting commands for the new configuration as soon as it is proposed, and without waiting for it to be agreed upon by the parent RSM. The proposals associated with the new configuration proceed to be ordered concurrently with the reconfiguration agreement, and are added to the trunk as soon as the configuration is agreed upon. This way, our framework allows unbounded degree of parallelism in the command execution during the transition periods avoiding the performance problems of the “brick-wall” solutions. In addition, the benefits of speculation become more substantial as the network delay grows thus making speculative solutions attractive in wide-area network settings.

The modularity of our framework enables a range of additional features useful in practical settings. One such feature is supporting rolling software upgrades: i.e., the implementation of the deployed RSMs can be replaced with the newer one without stopping the system. Likewise, misbehaving or buggy RSMs can be restarted or replaced on-the-fly with the minimum impact on the system operation as per the recovery-oriented computing (ROC) [19] guidelines.

We used our framework to implement a full-fledged reconfigurable replication platform using non-reconfigurable Paxos [12, 13] as its underlying non-reconfigurable RSM, and experimentally studied its performance. The results demonstrate that our system achieves high throughput and low latency in the absence of reconfigurations, which stay almost unchanged under highly dynamic reconfiguration scenarios. Specifically, the throughput is unaffected in the runs with reconfiguration rate of 5 per second, and degrades only by 20% when reconfiguration rate achieves that of 20 per second. In addition, our study indicates that the command latency in the vicinity of reconfiguration stays the same as that in the absence thereof.

In summary, the contributions made by our work are as follows:

- new generic framework for constructing reconfigurable RSM from non-reconfigurable ones, which is simple, modular, and efficient;
- new speculative approach to enable overlapping command ordering with the reconfiguration protocol thus reducing the reconfiguration latencies;
- practical implementation that leverages our framework to transform Paxos-based RSM to a highly efficient reconfigurable RSM;
- detailed experimental study of the implementation above demonstrating the benefits of our framework in practical settings.

The rest of the paper is organized as follows. Related work is discussed in Section 2. System model is presented in Section 3. Section 4 introduces non-reconfigurable and reconfigurable variants of RSM, and rigorously specify their properties. In Section 5, we present our reduction algorithm, and argue its correctness. The replication platform built on top of our framework, and its performance are discussed in Section 6. Section 7 concludes the paper.

2 Related work

Modular approaches to specifying RSM and their constituent building blocks have been addressed in several prior works. Boichat et al. [5] introduced modular decomposition of Paxos into weak leader election and round-based consensus abstraction, and [7] studied the abstraction of ranked register capturing the essence of the Paxos consensus algorithm. These papers however, do not address reconfiguration. Stoppable Paxos [10] presents a framework for combining non-reconfigurable Paxos instances into a reconfigurable RSM, but does not attempt to abstract away the internals of the underlying Paxos implementation. In addition, [10] does not support concurrent execution of the individual Paxos instances.

Several approaches to alleviating reconfiguration bottleneck in reconfigurable state machines have been proposed. The original idea by Lamport, described in [12, 13, 14], and implemented in SMART [16], was to delay the effect of the configuration agreed in a specific consensus instance by a fixed number α of successive consensus instances. If the configuration must take effect immediately, the remaining instances can be skipped by passing a special “window closure” decree consisting of α consecutive *noop* instances. Although this approach allows up to α consecutive commands to be executed concurrently, choosing the

right value of α is nontrivial. On the one hand, choosing α to be too small may under-utilize the available resources. On the other hand, large values of α may not match the actual service reconfiguration rate resulting in too frequent invocations of the window closure decrees (which must complete synchronously).

Chubby [6] and ZooKeeper [9] expose high-level synchronization primitives (respectively, locks and watches) that can be used to implement a reconfigurable state machine within the client groups. The solutions based on this approach are however, vulnerable to timing failures, and therefore, either restrict their failure model [25], or rely on additional synchronization protocols within the replication groups themselves to maintain consistency [15, 21].

Vertical Paxos [15] removes the configuration agreement overhead from the critical path by delegating it to an auxiliary “configuration master”. The reconfiguration involves an extra step of synchronizing with the read quorums of all preceding configurations causing throughput degradation. In addition, the configuration master itself is implemented using the α -based reconfigurable Paxos protocol, and therefore, suffers from the limitations similar to those discussed above.

Dynamic reconfiguration has been extensively studied in the context of virtually synchronous group communication [22, 4, 8, 20], and reconfigurable read/write registers [17, 1]. The reconfiguration protocols described in these papers do not aim to support consistency semantics as strong as those of state machine replication, and therefore do not directly apply in our context. Birman et al. [3] present a replication framework unifying reconfigurable state machine and virtual synchrony in which the normal operation is suspended during the reconfiguration periods.

Optimistic and speculative approaches to mask the coordination latency have been extensively studied in the past in a variety of contexts (such as e.g., group communication [23], and database replication [2]). However, to the best of our knowledge, speculative reconfigurable state machine replication studied in this paper has not yet been addressed in the prior work.

3 System Model

We consider an asynchronous message-passing system consisting of the (possibly infinite) set of processes P . Each pair of processes is connected by a point-to-point FIFO channel. For simplicity, we assume the *crash* failure model: i.e., the process experiencing a crash failure stops executing any further instructions forever. A process that never crashes throughout the system execution is called *correct*. The channels are reliable in the sense that a message sent by a correct process p to another correct process q is eventually received by q . Extending our techniques to support stronger benign failure models (such as crash/recovery, message omission, and network partitions and merges) is straightforward, and our prototype implementation described in Section 6 is capable of doing that.

4 State Machine Replication

In this section, we introduce the notions of non-reconfigurable and reconfigurable *Replicated State Machines (RSM)*, and specify their properties. Our specification style loosely follows the I/O automata formalism of [18].

For the following we will fix *Command* to be the set of the user commands, and *Config* be the set of the configuration identifiers. Each configuration C is associated with a finite set of processes in P , denoted $members(C)$.

Inputs: $join_{C,p}$ $propose(cmd)_{C,p}, cmd \in Command$ Outputs: $learn(cmd)_{C,p}, cmd \in Command$

Figure 1: The $NR-RSM(C)_p, C \in Config$, signature for process $p \in members(C)$:

4.1 Non-Reconfigurable RSM (NR-RSM)

The *Non-Reconfigurable Replicated State Machine* (NR-RSM) is parametrized by a configuration $C \in Config$. The interface supported by $NR-RSM(C)$ for each process $p \in members(C)$ appears in Figure 4.1. $NR-RSM(C)$ accepts the commands $cmd \in Command$ submitted by the C 's members through the $propose(cmd)_C$, requests and outputs a totally ordered sequence of $learn(cmd)_C$ notifications.

The process p 's instance of $NR-RSM(C)$, denoted $NR-RSM(C)_p$, is activated by the $join_{C,p}$ request. $NR-RSM(C)$ becomes operational once sufficiently many members p of C have executed the $join_{C,p}$ request. The precise initial membership depends on the NR-RSM implementation being used, desired resiliency, failure model, and environment properties. For example, if NR-RSM is implemented using the non-reconfigurable Paxos algorithm [13], $NR-RSM(C)$ may become operational as soon as a majority of $members(C)$ have executed $join_{C,p}$. We do not explicitly model the process "leaves" since they are equivalent to the process crashes in the crash failure model.

We now specify the set of properties that must be satisfied by the correct implementations of NR-RSM. The properties are defined in terms of the sequences of actions α of its external interface in Figure 4.1. We start by specifying the environment assumptions that must hold in order for the NR-RSM implementation to be correct:

Property 1. (Well-Formedness)

- If $e = propose_{C,p}$ is an event in α , then there exists $join_{C,p}$ that precedes e in α .
- For each process $p \in members(C)$, at most one $join_{C,p}$ occurs in α .
- For each command $cmd \in Command$, at most one $propose(cmd)_{*,*}$ occurs in α .

Next, we specify the $NR-RSM(C)$ safety properties. The Integrity property below asserts the following two facts: (1) a process cannot output any $learn$ events before it has joined $NR-RSM(C)$, and (2) each learnt command must be previously proposed. Formally,

Property 2 (Integrity). *Let $e = learn(cmd)_{C,p}$ be an event in α . Then, the following holds:*

- There exists $join_{C,p}$ event that precedes e in α .
- There exists $propose(cmd)_{C,q}$ event that precedes e in α .

The next property states that each command is learnt at most once by each process:

Property 3 (No Duplication). *For each command $cmd \in Command$ and process p , there exists at most one $learn(cmd)_p$ event in α .*

The following property requires that the commands are learnt in the same order by all members of C . Formally:

Property 4 (Linearizability). *There exists a sequence of commands $\bar{x} = x_1, x_2, \dots$ such that for each process $p \in \text{members}(C)$, if $\bar{\pi} = \pi_1, \pi_2, \dots$ is the sequence of the learn_p events output by p in α , then, for all $i \geq 1$, $\pi_i = \text{learn}(x_i)_{C,p}$.*

We now turn to specifying the NR-RSM liveness. The property below asserts that after some time t (which is the parameter of the property), each command proposed by a correct member of C is eventually learnt by all correct members of C provided they have joined NR-RSM(C). Formally, NR-RSM is said to be *live* after time t in α if the following holds:

Property 5 (Liveness). *If $\text{propose}(\text{cmd})_p$ is an event occurring at a correct process p after t , then for each correct process $q \in \text{members}(C)$ such that $\text{join}_{C,q}$ is an event in α , there exists $\text{learn}(\text{cmd})_{C,q}$ that occurs at time $t' > t$ in α .*

4.2 Reconfigurable RSM (R-RSM)

Inputs:
$\text{propose}(C, \text{cmd})_p, C \in \text{Config}, \text{cmd} \in \text{Command}, p \in \text{members}(C)$
$\text{recon}(C, C')_p, C, C' \in \text{Config}, p \in \text{members}(C)$
Outputs:
$\text{learn}(\text{cmd})_p, \text{cmd} \in \text{Command}$
$\text{new-conf}(C)_p, C \in \text{Config}$
$\text{ready}(C)_p, C \in \text{Config}, p \in \text{members}(C)$

Figure 2: The R-RSM Signature for Process $p \in P$:

The R-RSM external interface appears in Figure 4.2. In contrast to the non-reconfigurable RSM, the configuration of *Reconfigurable RSM* (R-RSM) can be changed dynamically through the user inputs. Initially, the R-RSM's configuration is the distinguished configuration C_0 . Afterwards, new configurations are proposed through the $\text{recon}(C, C')$ requests, which include two parameters: a known configuration C , and the target configuration C' to which C is requested to reconfigure.

The recon requests can only be initiated at one of the members of C . The members of C' respond with $\text{ready}(C')$ events once they are aware of C' , and ready to accept proposals associated with C' . If C' is eventually accepted as the configuration to follow C in the global order, the members q of both C and all of the subsequent configurations (including C') will be notified of that through the $\text{new-conf}(C')_q$ events.

Similarly to NR-RSM, R-RSM accepts commands submitted through the propose requests, and outputs learn notifications for the commands whose slot in the total order has been finalized. In addition, each propose request also includes the configuration parameter that indicates the desired configuration to be used to order the provided command.

Note that the R-RSM interface allows the proposals (of both commands and configurations) to be issued against configurations, which have been reported as ready, but whose ordering has not yet been completed (and may subsequently fail). This provides implementations with flexibility in choosing how to deal with the proposals that have been received by R-RSM while reconfiguration is still in progress. In our implementation (see Section 5), these proposals are ordered *speculatively* using a separate instance of the non-reconfigurable

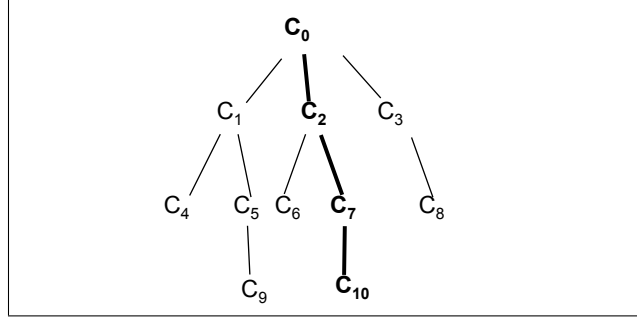


Figure 3: The configuration tree formed by the *recon* events. The trunk configurations are highlighted.

state machine being executed concurrently with the reconfiguration protocol. As we show in Section 6, this leads to substantial reductions in the reconfiguration delays provided the proposed configuration will be eventually included into the total order (which is a common occurrence in practice). The implementation can limit or completely eliminate speculative behavior by tweaking the timing of the *ready* reports.

All configurations submitted in the course of the R-RSM execution α form the *tree* with the root C_0 , and C being the parent of C' for all $C, C' \in \text{Config}$ iff $\text{recon}(C, C')$ has occurred in α (see Figure 3). The configurations reported in the *new-conf* notifications form a continuous path, called the *trunk*, in the configuration tree starting from C_0 .

As before, we specify the correctness properties for the R-RSM implementation in terms of the sequences of actions α of its external interface in Figure 4.2. We start by defining the well-formedness conditions, which capture the permissible interaction patterns between the R-RSM and its environment:

Property 6. (Well-Formedness)

- If $e \in \{\text{propose}(C, *)_p, \text{recon}(C, *)_p\}$ is an event in α , then there exists $\text{ready}(C)_p$ that precedes e in α unless $C = C_0$
- For each command $\text{cmd} \in \text{Command}$, there exists at most one $\text{propose}(\text{cmd})_*$ in α .
- For each configuration $C \in \text{Config}$, there exists at most one $\text{recon}(*, C)_*$ in α ¹.

Next, we specify the R-RSM safety properties. The Integrity property below asserts that each *learn* (resp., *new-conf*) notification delivered at a process p must be preceded by both: (1) p 's join, and (2) $\text{propose}(C, *)$ ($\text{recon}(C, *)$) where C is the latest configuration that has been delivered at p before the notification event. Formally:

Property 7 (Integrity). *Let $e \in \{\text{learn}(\text{cmd})_p, \text{new-conf}(C')_p, \text{ready}(C')_p\}$, be an event in α , and C be the configuration delivered by the latest *new-conf* event that precedes e at p . Then, all of the following holds:*

- If $e = \text{learn}(\text{cmd})_p$, then there exists $q \in \text{members}(C)$ such that $\text{propose}(C, \text{cmd})_q$ (resp., $\text{recon}(C, C')_q$) is an event in α .
- If $e \in \{\text{new-conf}(C')_p, \text{ready}(C')_p\}$, then there exists $q \in \text{members}(C)$ such that $\text{recon}(C, C')_q$ is an event in α .

¹Note that only the proposed configuration *identifiers* are required to be unique, not the set of members associated with those identifiers.

The next property states that each command (resp., configuration) is learnt (resp., delivered) at most once by each process:

Property 8 (No Duplication). *For each $cmd \in Command$, there exists at most one $learn(cmd)_p$, and for each $C \in Config$, there is at most one $ready(C)_p$, and at most one $new-conf(C)_p$ event that occurs at each process $p \in P$ in α .*

The most important safety property supported by R-RSM is *linearizability*, which requires that all processes deliver the same sequence of *learn* and *new-conf* events. Formally,

Property 9 (Linearizability). *There exists a sequence of commands and configurations $\bar{x} = x_1, x_2, \dots$ such that for each process $p \in P$, if $\bar{\pi} = \pi_1, \pi_2, \dots$ is the sequence of the $learn_p$ and $new-conf_p$ events output by p in α , then for all $i \geq 1$, if $x_i \in Config$, then $\pi_i = new-conf(x_i)_p$, and if $x_i \in Command$, then $\pi_i = learn(x_i)_p$.*

We now specify the R-RSM liveness. The following property says that if every proposed configuration includes at least one correct member, then eventually, the system stabilizes in the sense that (1) each command proposed by a correct process is eventually learnt provided new reconfigurations attempts cease to be initiated, and (2) each proposal to reconfigure a previously installed configuration eventually triggers a *new-conf* event. Formally,

Property 10 (Liveness). *Suppose that every configuration proposed in α includes at least one correct member. Then, there exists Global Stabilization Time (GST) such that for all times $t > GST$, if C_0, C_1, \dots, C_k is the sequence of configurations induced by the longest sequence of *new-conf* events delivered by some process before t , then, both of the following holds:*

- *If no $recon(C, *)$ events with $C = C_k$ occur after GST, then for each correct $p \in members(C_k)$, if $propose(C_k, cmd)_p$ is an event that occurs after t in α , then it is eventually followed by $learn(cmd)_q$ at all correct $q \in members(C_k)$.*
- *There exists configuration C such that if a correct $p \in members(C_k)$ invokes $recon(C_k, *)_p$ after t , then it is eventually followed by $new-conf(C)_q$ at all correct $q \in members(C_k) \cup members(C)$.*

In addition, we require that each $recon(*, C)$ invoked by a correct process is eventually followed by the $ready(C)$ notification delivered at each correct member of C .

Property 11. *If $recon(*, C)_p$ is an event that occurs at a correct process p in α , then it is followed by $ready(C)_q$ for each correct $q \in members(C)$.*

5 Reduction Algorithm

In this section, we present the implementation of the algorithm to transform a collection of NR-RSM(C), $C \in Config$ to R-RSM, and argue its correctness. The implementation is the composition of the automata R-RSM $_p$ for each process $p \in P$. The code executed by each R-RSM $_p$ appears in Algorithm 1.

The implementation of R-RSM can be viewed as consisting of the following three phases:

Configuration-specific ordering: Each proposed configuration C is associated with the non-reconfigurable state machine NR-RSM(C), which is used to order the commands and configurations associated with C . The local outputs of each NR-RSM(C) $_p$ are stored in the *branches $_p$* data structure (line 1.7), which maps each C to the sequence of commands and configurations output by $learn_{C,p}$ events.

Algorithm 1 R-RSM from $\{\text{NR-RSM}(C)\}$. The R-RSM_p code.

```

1: Types
2:   Config: the set of configuration identifiers with the initial identifier  $C_0$ 
3:   Command: the set of command identifiers
4: var
5:   status  $\in \{\text{active}, \text{idle}\}$ 
6:   view  $\subseteq P$ 
7:   branches: mapping from Config to  $\text{seqOf}(\text{Config} \cup \text{Command}) \cup \{\perp\}$ 
8:   trunk:  $\text{seqOf}(\text{Config} \cup \text{Command})$ 
9:   next  $\in \mathbb{N}^{\geq 0}$ 
10:  curConf  $\in \text{Config}$ 
11: Initialisation:
12:  branches( $C$ )  $\leftarrow \perp$  for all  $C \in \text{Config}$ 
13:  trunk  $\leftarrow []$ 
14:  next  $\leftarrow 0$ 
15:  curConf  $\leftarrow C_0$ 
16:  view  $\leftarrow \emptyset$ 
17:  if  $p \in \text{members}(C_0)$  then
18:    joinConf( $C_0$ )
19:    view  $\leftarrow \text{members}(C_0)$ 
20:  end if
21: propose( $C, \text{cmd}$ ):
22:  if branches( $C$ )  $\neq \perp \wedge \text{branches}(C)[i] \notin \text{Config}$  for all  $0 \leq i \leq \text{length}(\text{branches}(C))$  then
23:    propose( $\text{cmd}$ ) $C$ 
24:  end if
25: recon( $C, C'$ ):
26:  if branches( $C$ )  $\neq \perp \wedge \text{branches}(C)[i] \notin \text{Config}$  for all  $0 \leq i \leq \text{length}(\text{branches}(C))$  then
27:    for all  $q \in \text{members}(C')$  do
28:      send( $\langle \text{JOIN}, C, C' \rangle$ ) $p, q$ 
29:    end for
30:    propose( $C'$ ) $C$ 
31:    view  $\leftarrow \text{view} \cup \{C'\}$ 
32:  end if
33: upon learn( $x$ ) $C$ : do
34:  Append  $x$  to branches( $C$ )
35:  if  $C = \text{curConf} \wedge \text{next} < \text{length}(\text{branches}(\text{curConf}))$  then
36:    learnNext(branches(curConf)[next])
37:  end if
38: end upon
39: upon receiving  $\langle \text{STATE}, tr, v \rangle$  message from  $q \in P$  do
40:   for all  $i = \text{length}(\text{trunk})$  to  $\text{length}(tr) - 1$  do
41:     learnNext( $tr[i]$ )
42:   end for
43:   view  $\leftarrow \text{view} \cup v \cup \{q\}$ 
44: end upon
45: upon receiving  $\langle \text{JOIN}, C, C' \rangle$  message from  $q \in \text{members}(C)$  do
46:   if branches( $C'$ )  $= \perp$  then
47:     joinConf( $C'$ )
48:     Output ready( $C'$ )
49:     view  $\leftarrow \text{view} \cup \text{members}(C) \cup \text{members}(C')$ 
50:   end if
51: end upon
52: task State Transfer
53:   if status  $= \text{active}$  then
54:     Periodically:
55:       for all  $q \in \text{view}$  do
56:         send( $\langle \text{STATE}, \text{trunk}, \text{view} \rangle$ ) $p, q$ 
57:       end for
58:   end if
59: end
60: procedure learnNext( $x$ ),  $x \in \text{Command} \cup \text{Config}$ 
61:  Append  $x$  to trunk
62:  if  $x \in \text{Config}$  then
63:    if branches( $x$ )  $= \perp \wedge p \in \text{members}(x)$  then
64:      joinConf( $x$ )
65:      Output ready( $C$ )
66:      view  $\leftarrow \text{view} \cup \text{members}(x)$ 
67:    end if
68:    Output new-conf( $x$ )
69:    next  $\leftarrow 0$ 
70:    curConf  $\leftarrow x$ 
71:  else
72:    Output learn( $x$ )
73:    next  $\leftarrow \text{next} + 1$ 
74:  end if
75: end
76: procedure joinConf( $C$ ),  $C \in \text{Config}$ 
77:  if status  $= \text{idle}$  then
78:    status  $\leftarrow \text{active}$ 
79:  end if
80:  branches( $C$ )  $\leftarrow []$ 
81:  join $C$ 
82: end

```

Configuration tree pruning: The branches of the configuration tree in Figure 3 are *pruned* so that the surviving configurations form a single consistent *trunk*. To ensure that all processes prune the branches in a consistent fashion, the successor C' of each configuration C is determined by the output of $\text{NR-RSM}(C)$.

Total order construction: Each process p concatenates the totally ordered fragments stored in branches_p in the order induced by the configuration trunk to produce a globally-consistent sequence of commands and configurations, which is stored in the trunk_p variable. trunk_p is then iterated to produce a globally consistent sequence of the *learn* and *new-conf* outputs.

The above three phases are executed concurrently, with the configuration-specific ordering phase being executed concurrently for each individual configuration. Below we describe the implementation of each of the above phases in more detail.

Configuration-Specific Ordering: A process p joins $\text{NR-RSM}(C)$ when it either receives the $\langle \text{JOIN}, C \rangle$ message (lines 1.46–1.50), or encounters C in the course of the total order construction phase (lines 1.63–1.67). From that point on, all commands and configurations x submitted by process p through either $\text{propose}(C, x)_p$ or $\text{recon}(C, x)_p$ are forwarded to the p 's local instance of $\text{NR-RSM}(C)$ through the $\text{propose}(x)_{C,p}$ requests (see lines 1.23 and 1.23). Each ordered command or configuration delivered by the $\text{learn}_{C,p}$ event is then appended to $\text{branches}(C)_p$.

The state machines created upon the reception of the JOIN messages are used to *speculatively order* the proposals corresponding to the configurations, which have not yet been incorporated into the global total order, thus reducing the reconfiguration time. We discuss speculation in more detail in Section 5.3 below.

Configuration Tree Pruning: The configuration tree is pruned by selecting the *first* configuration appearing in $\text{branches}(C)_p$ to be the C 's successor by all processes $p \in \text{members}(C)$. Since all entries of $\text{branches}(C)_p$ appear in the same order at all processes $p \in \text{members}(C)$, they all will choose the same configuration C' as the C 's successor. All other processes will learn of C' through state transfer (see below). The successor chosen for each configuration C becomes explicit when it is incorporated into trunk_p in the course of the total order construction phase (lines 1.68–1.67).

Total Order Construction: The chain of the surviving branches currently known to each process p is kept in the trunk_p variable (see line 1.8). In addition, curConf_p (see line 1.10) holds the configuration corresponding to the last branch on trunk_p . Initially, $\text{curConf}_p = C_0$ (line 1.15) so that trunk_p consists of a single branch corresponding to the root of the configuration tree C_0 .

As the entries are added to $\text{branches}(\text{curConf}_p)$ (lines 1.34–1.37), they are copied to trunk_p (line 1.61) and learnt one-by-one (line 1.72–1.73) until a new configuration C is encountered. At this point, curConf_p is reassigned to C thus choosing $\text{branches}(\text{curConf}_p)$ as the next branch to feed trunk_p (lines 1.68–1.67).

Since as we argued above, C is the only configuration that can be chosen as the curConf_p 's successor at all processes $q \in P$, $\text{branches}(C)$ can be the only one selected to follow $\text{branches}(\text{curConf}_q)$ in trunk_q . We therefore, proved the following:

Lemma 1. *All well-formed executions of the R-RSM implementation in Algorithm 1 satisfy Property 9*

5.1 State Propagation

Since for each configuration C , the commands and configurations ordered by $\text{NR-RSM}(C)$ are only delivered to the members of C , we need an additional mechanism to ensure they propagate down the configuration trunk. This mechanism is implemented by the *State Transfer* task (lines 1.52–1.59), which is executed in the background at each process p that have joined at least one of the proposed configurations.

It proceeds by continuously gossiping its entire trunk_p (line 1.56) to the set of processes being accumulated in the view_p variable². By line 1.66, view_p is guaranteed to include the members of all configurations comprising the configuration trunk known to p . In particular, this includes the members of the configuration

²Note that in reality the state transfer can be made considerably more efficient through a variety of techniques, such as e.g., push-pull gossip[24], and log contraction [14]

C' that immediately follows C in $trunk_p$. If p is correct, all correct members of C' will eventually receive $trunk_p$ that will include the entire sequence of commands ordered within C up to and inclusively of C' . Moreover, once there is a correct $q \in members(C')$ that delivers $new-conf(C')$, all correct members of both C' and its immediate successor (if any) are guaranteed to receive the entire trunk known to the members of C even if all members of C stop gossiping their trunks. We therefore, have the following:

Lemma 2 (State Propagation). *Let C and C' be two configurations such that C' follows C in the configuration trunk. Then, if there exists a correct member p of C that continues to gossip $trunk_p$ until $new-conf(C')$ is delivered by a correct member q of C' , then all correct members of C' will eventually incorporate all commands and configurations ordered within C into their trunks.*

The result above establishes criteria that can be used in practice for determining when the members of old configurations can be taken off line without compromising the system liveness. For example, suppose that a majority of the members of each installed configuration do not crash, unless they are taken off line in an orderly fashion. Then, for each configuration C , a minority of $members(C)$ can be taken off line once a majority of $members(C)$ deliver $new-conf(C')$ notification for some configuration C' ; and the remaining members can be disconnected once a majority of C' delivers $new-conf(C')$.

5.2 Liveness

The liveness of our implementation follows from the liveness of its constituent non-reconfigurable state machines and the State Propagation property above.

Specifically, assume that all state machines instantiated in the course of the execution of the code in Algorithm 1 become live (in the sense of Property 5) after some time t . Suppose that C is the last configuration of the system. Since all configurations include at least one correct process, by Lemma 2, there is a time $t' > t$ such that all correct members of C will incorporate C into their trunks and join $NR-RSM(C)$ before t' .

If no new reconfiguration attempts are made from this point onwards, every command cmd proposed by a correct member of C through $propose(C, cmd)$ is guaranteed to appear in $branches(C)_q$, followed by $trunk_q$, until it eventually output through $learn(cmd)_q$. Otherwise, there will be C' such that $recon(C, C')$ will result in C' to be included into $trunk_p$ of all correct processes in C , from where by Lemma 2, it is guaranteed to eventually propagate to all correct members of C' . Hence, we proved the following:

Lemma 3. *Each well-formed execution α of the R-RSM implementation in Algorithm 1 satisfies Property 10 provided all NR-RSM implementations instantiated in α are live after some t .*

5.3 Speculation

One important property of our R-RSM implementation is that the non-reconfigurable state machine for each configuration C is completely *independent* from the state machines for other configurations, and could proceed ordering the commands and configurations submitted under C *concurrently* with them. In particular, since the members of C are notified of the new configuration immediately upon receiving its corresponding reconfiguration request (line 1.27– 1.29), this also applies to the commands submitted under C while the successor of its parent configuration is still being chosen. These commands will be available for incorporating into the trunk and subsequent learning as soon as C becomes the chosen successor of its parent configuration, thus reducing (or eliminating altogether) reconfiguration delays.

The precise degree of the performance improvement depends on the choice of the underlying non-reconfigurable state machine implementation, and the network parameters (primarily, the communication

delay). As we show in Section 6, for the Paxos-based R-RSM implementation, speculation helps to eliminate penalties associated with throughput and latency during reconfiguration times, making them almost indistinguishable from those achieved during the normal operation.

6 Implementation and Evaluation

We applied our reduction framework to implement a full-fledged reconfigurable replication platform using non-reconfigurable Paxos [12, 13] as its underlying NR-RSM implementation.

To shield the users from intricacies of selecting viable configurations, our replication platform augments R-RSM with two additional modules: Configuration Manager (CM) and Command Queue (CQ). CM implements the logic to detect failures, and produce new configurations, which are then submitted to R-RSM through the *recon* requests. The configuration membership is based on the failure suspicions and user policies. An example policy may ask for every configuration to include sufficiently many healthy nodes. CQ is responsible for associating user commands with configurations and submitting them to the platform through the *propose* requests. The configurations are chosen based on the *ready* notifications thus leveraging the speculation capabilities of R-RSM to reduce reconfiguration delays.

Each Paxos instance created by R-RSM is supplied with the identifier of a deterministically chosen leader thus avoiding the overheads of the first phase of Paxos if the configured leader does not fail. Thus, normally, only the second phase of Paxos is needed to complete each individual agreement instance resulting in a single round-trip delay to order a command.

We studied the performance of our implementation experimentally using the testbed comprised of 4 IBM HS22 blades equipped with Intel Xeon X5670 processor with 24 2.93GHz cores and 64GB RAM. Each machine was equipped with 1GB network card, and ran Red Hat Linux. A single replica was hosted on each machine.

The replicas were subjected to request streams generated in either *synchronous* or *asynchronous* fashion. In the synchronous mode, the requests were issued from multiple threads each of which was waiting for the response to the previously submitted request to arrive before submitting the new one. The synchronous mode allowed us to exercise high degree of control over the offered system load by varying the number of simultaneously executed synchronous threads. In contrast, in the asynchronous mode, each thread was submitting requests as they were generated without waiting for the prior requests completion. The asynchronous mode was used to drive the system to its maximum utilization.

In the first experiment, we studied the throughput and latency as a function of the offered system load in the absence of reconfiguration using a single configuration consisting of 3 replicas. The results (see Figure 4) show that our system is capable of achieving the sustained throughput of 12K request/second before the latency starts to rapidly grow.

In the next experiment, we studied the impact of our speculative reconfiguration mechanism on the command throughput. For that, the system was subjected to a series of reconfiguration requests submitted at varied frequency. The normal commands were submitted using 10 synchronous threads, which corresponds to the maximum sustained system load of 12K requests/second (see Figure 4). The measurements are depicted (see Figure 5) as absolute command throughput in the presence of reconfiguration, and percentage of degradation relative to the maximum sustained throughput (12K). The results show that the throughput in the presence of reconfigurations is almost identical to that achieved in the absence thereof, reaching maximum 20% of degradation when the system is reconfigured 20 times/second.

In the last experiment, we studied the degree of improvement produced by speculation in terms of the latencies of the commands submitted in close proximity of the reconfiguration requests. The results (see

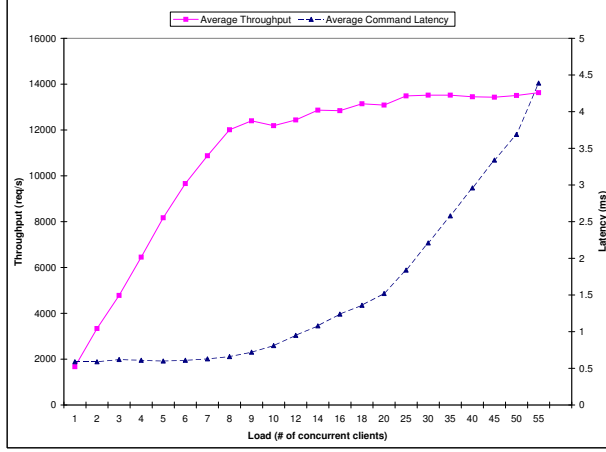


Figure 4: Throughput and Latency in the Absence of Reconfiguration.

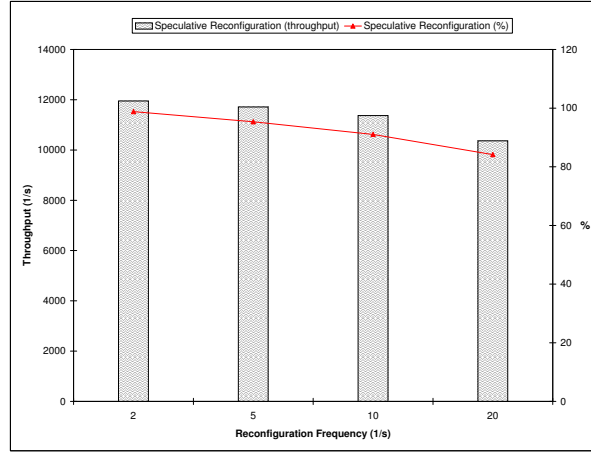


Figure 5: Throughput With Varied Reconfiguration Frequency.

Figure 6) indicate that with speculation, the latency is unaffected by reconfiguration staying closely to that of the normal mode (i.e., in the absence of reconfiguration) throughout the entire reconfiguration period. In contrast, without speculation, the latency increases sharply for the first command, and then keeps decreasing slowly until reaching the normal mode value in the proximity of the 100th command.

7 Conclusions

We have introduced a modular framework for transforming a collection of arbitrary non-reconfigurable replicated state machines (RSM) into the reconfigurable RSM implementation. Our framework follows the *black box* approach, and does not make any assumptions with respect to its execution environment apart from reliable channels. The individual state machines instantiated by our implementation can be executed concurrently to one another, in particular, overlap each other execution in a *speculative* fashion to mask the reconfiguration delays. We have applied our framework to build a prototype of an end-to-end dynamic replication platform using non-reconfigurable Paxos as its underlying building block, and studied

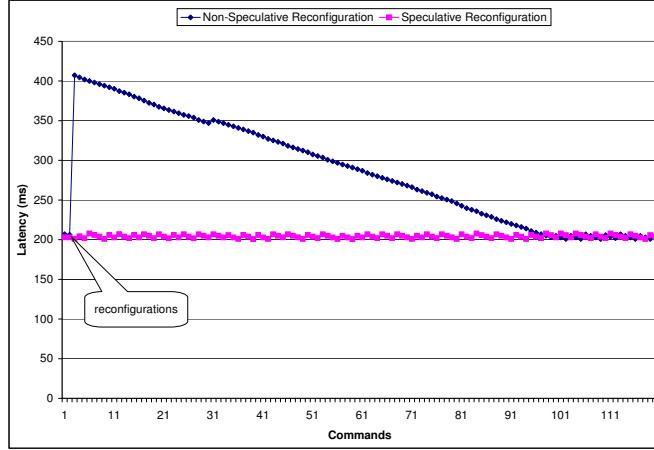


Figure 6: Command Latency in the Vicinity of Reconfiguration. The network delay was simulated to be 100ms on average.

its performance on a realistic distributed testbed. Our results demonstrated that our platform achieves high throughput and low latency in the absence of reconfigurations, which stay almost unchanged under highly dynamic reconfiguration scenarios.

Our platform is being developed at IBM Research, and slated to be included into the future IBM's platform-as-a-service offerings as a foundational tool. In the future, we intend to explore the ways to extend our speculation-based approach to support optimistic replication in wide-area network settings, and augment it with an on-line optimization framework to facilitate selection of configurations based on the changing network parameters.

References

- [1] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.
- [2] Y. Amir and C. Tutu. From total order to database replication. In *ICDCS*, pages 494–, 2002.
- [3] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.
- [4] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, pages 123–138, 1987.
- [5] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.
- [6] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
- [7] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.
- [8] D. Dolev, I. Keidar, and E. Y. Lotem. Dynamic voting for consistent primary components. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 63–71, New York, NY, USA, 1997. ACM.
- [9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [10] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=101826>, 2008.
- [11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.
- [12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [13] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.
- [14] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. Technical report, Microsoft Research, 2008.
- [15] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, New York, NY, USA, 2009. ACM.
- [16] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *In Proc. EuroSys06*, 2006.

- [17] N. Lynch and A. A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *In DISC*, pages 173–190, 2002.
- [18] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
- [19] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, 2002.
- [20] R. D. Prisco, A. Fekete, N. A. Lynch, and A. A. Shvartsman. A dynamic primary configuration group communication service. In *DISC*, pages 64–78, 1999.
- [21] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4:243–254, January 2011.
- [22] A. Ricciardi and K. Birman. Process Membership in Asynchronous Environments. Technical Report TR93-1328, Cornell University, February 1993.
- [23] J. Sussman and K. Marzullo. The bancomat problem: an example of resource allocation in a partitionable asynchronous system. *Theor. Comput. Sci.*, 291:103–131, January 2003.
- [24] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Large-Scale Distributed Systems and Middleware (LADIS 2008)*, September 2008.
- [25] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.